Optimizing Graph Queries with Graph Joins and Sprinkle SPARQL

Eric L. Goodman, Edward Jimenez
Sandia National Laboratories
Albuquerque, NM, USA

Sinan al-Saffar
Semantic Scale LLC,
Tampa, FL, USA

Cliff Joslyn, David Haglin
Pacific Northwest National Laboratory
Richland, WA, USA

Dirk Grunwald
University of Colorado, Boulder, USA

***Abstract*—Big data problems are often more akin to sparse graphs rather than relational tables. As such we argue that graph-based physical representations provide advantages in terms of both size and speed for executing queries. Drawing from research in sparse matrices, we use a compressed sparse row (CSR) format to model graph-oriented data. We also present two novel mechanisms for exploiting the CSR format that both find optimal join strategies and also prune variable bindings before expensive join operations occur. The first tactic we call Sprinkle SPARQL, which takes triple patterns of SPARQL queries and performs low-cost, linear-time set intersections to produce a constrained list of variable bindings for each variable in a query. Besides constrained lists of variable bindings, Sprinkle SPARQL also produces metrics that are consumed by the join algorithm to select an optimal execution path. The second tactic, graph joins, utilizes the CSR data structure as an index to efficiently join two variables expressed in a triple pattern together. We evaluate our approach on two data sets with over a billion edges: LUBM(8000) and an R-MAT graph generated with Graph500[1] parameters and extended to have edge labels.**

I. Introduction

Big Data often comes in the form of large graphs, where entities or nodes are interconnected with edges. Notable examples include social networks, the internet itself with hyperlinks forming edges, and biological networks such as protein-protein interaction networks. In this paper we examine the Semantic Web, which is a growing body of data: Linked Data on the web is over 50 billion edges.

The Semantic Web has several standards. The Resource Description Framework (RDF)[2] describes the data. RDF data consists of lists of triples, a *subject*, *predicate*, and *object*, where the subject and object are nodes and the predicate is a labeled edge between them. SPARQL[3] is for querying.

In this paper, we analyze and evaluate empirically a new algorithm for performing SPARQL Queries that we call *Sprinkle SPARQL*. Our key contribution in this paper is to show that Sprinkle SPARQL fulfills two desiderata for query engines, namely 1) removing RDF terms[4] from consideration prior to executing a query, and 2) selection of a near optimal execution plan.

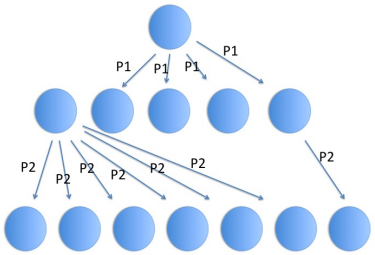


Figure 1. An example graph showing how the ordering of evaluating triple patterns is important. Also shows that even with the best ordering, work is wasted on intermediate results that are later pruned.

To understand each point, consider the graph in Figure 1 as representing the search space and suppose we have the following query: `SELECT ?X1 ?X2 ?X3 WHERE {?X1 p1 ?X2 . ?X2 p2 ?X3}` By inspection, the optimal path is to first evaluate `?X1 p1 ?X2` followed by `?X2 p2 ?X3` requiring five edge examinations. The other order examines eight edges. However, even if we select the optimal path, for a naïve approach, 75% of the intermediate results created after evaluating `?X1 p1 ?X2` are later pruned from the final answer. More complex queries easily exacerbate these two situations.

Sprinkle SPARQL addresses these two issues, using low cost operations we call *Sprinkling* to prune dead-ends and to find a near optimal execution plan. The name stems from the fact that constraints from each triple pattern *sprinkle* or rain down onto associative arrays, one for each variable. For the data sets we examined, our approach does indeed select the

[1] http://www.graph500.org/
[2] http://www.w3.org/RDF/
[3] http://www.w3.org/TR/rdf-sparql-query/
[4] http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/#defn_RDFTerm

978-1-4799-5666-1/14/$31.00 ©2014 IEEE

optimal path for each query. Also, we trimmed intermediate results by up to 60%. We also introduce the notion of *Graph Joins*, which are similar to joins in the relational world but where we use a graph data structure to index the data in ways in which the data is expressed.

Borrowing terminology from the *Exascale Computing Study* [3], our work advances capability-driven graph database applications, where the focus is producing results quickly, rather than capacity-driven applications, where throughput of many low complexity problems is the key goal. We show scalability to 128 processors for more complicated queries.

II. Related Work

Almost all previous approaches to processing SPARQL queries focus on disk-based solutions, either with single node/single disk-type resources, which can conceivably be scaled in a task parallel manner, [1, 10, 11, 13], or with clusters using MapReduce [2, 8]. They make the case that data sizes are growing to the extent that the problems cannot be entirely stored in memory. We make an alternative argument, that there do exist systems with extensive shared memory on the order of many terabytes. While RDF systems based on disk storage are likely cheaper than memory resident approaches, at times performance trumps cost.

Another differentiation between our approach and previous ones is our focus on parallelizing individual queries. The single node/single disk-type approaches focus on throughput of large amounts of simple queries. While throughput is important to many applications, we are more interested in speeding up complicated queries. The MapReduce-based approaches do parallelize, but due to the writing back and forth to disk, our memory-resident approach is at least an order of magnitude faster, making it difficult to compare the two in any meaningful way.

III. Data Structures

A common approach, we encode the RDF triples into integers using a dictionary encoding scheme using an in-memory hash table to map from strings to integers [6, 5]. Besides the dictionary, there are two fundamental data structures used by the Sprinkle SPARQL and our graph join strategy: a common technique for sparse matrix representation called compressed sparse row (CSR) and a multimap. The CSR provides a compact data structure that gives the ability to look up a specific node in the graph and in time proportional to the outdegree (or indegree) of the node and find the outgoing (or incoming) edges of the node. The multimap provides a mapping from predicate type to triples with the given predicate. These data structures and other auxiliary structures are discussed in more detail in Figure 2.

IV. Our Approach

Evaluation of basic graph patterns consists of two phases:1) Sprinkle SPARQL and 2) Graph Joins. Sprinkle SPARQL evaluates each triple pattern in isolation, storing in a hash table a list of variable bindings that satisfy each triple pattern. While triple patterns are evaluated in isolation, known bindings from previous steps can be used in later steps. Sprinkle SPARQL creates a constrained list of bindings before expensive join operations. After creating a constrained list of bindings, they are then combined together using what we call graph joins. We discuss each phase below.

A. Sprinkle SPARQL

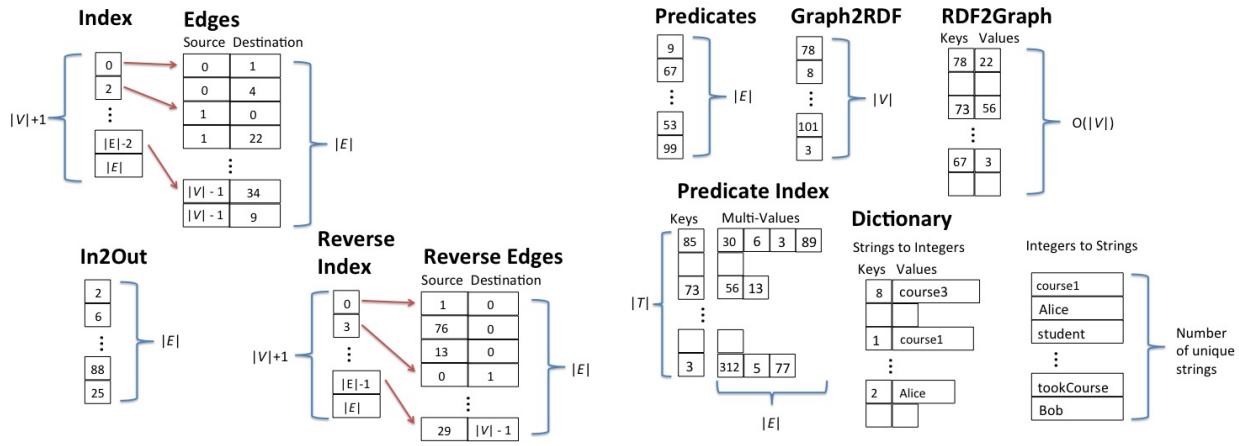
To impart some intuition of how Sprinkle SPARQL works, we start with an example. Consider the query (equivalent in intent to LUBM query 1) below:

```
SELECT ?X WHERE {?X tookCourse course1
. ?X isA gradStudent}
```

Remember the two data structures, the CSR graph and the multimap. The CSR tells us in constant time the in-degree of nodes *course1* and *gradStudent*. It is likely that $\delta^-(course1) \ll \delta^-(gradStudent)$, where $\delta^-(x)$ denotes the indegree of x. From the multimap we find in constant time the number of edges that have predicate value *tookCourse* and predicate value *isA*. Both of these counts are likely much larger than $\delta^-(course1)$. Thus, the algorithm will select to expand out from the in-edges of *course1*. This is shown in Figure 3(a). We create a hash table that is around two times the indegree of *course1* (for avoiding collisions). We then iterate over the edges of *course1*, checking to see that the predicate type is *tookCourse*, and if so add the source nodes to a hash table and increment the counter in the hash table. This hash table represents the valid variable bindings for variable *?X*. In the figure, the identifiers are strings for demonstration purposes, but in actuality everything at this point in the graph is integers.

We next evaluate the second triple pattern. As before, we can examine the counts for how many edges have predicate type *isA* and also the indegree of *gradStudent*. However, now that we have examined variable *?X*, we can also look at the total outdegree of the variable bindings in the hash table. Every time a hash table is updated after a sprinkle, we also update counts for total indegree and total outdegree of the nodes present therein. As shown in Figure 3(b), the algorithm chooses to expand from variable bindings in the hash table as the total out-edges for the variable bindings is much less than the indegree of *gradStudent* and the count of edges with label *isA*. Of course there will be wasted work as some of the out-edges of *Bob* and *Alice* are not of predicate type *isA*, but the total is much less than other options.

The general procedure for the Sprinkle Phase is to iterate over each triple pattern in a basic graph pattern, greedily



(a) CSR

(b) Other data structures

Figure 2. (a) presents the components of the compressed sparse row graph. The *Edges* array contains a list of all edges, sorted by source. *Index* provides offsets into this array for each vertex. *Reverse Edges* is the same list of edges, but now sorted by the destination. The *Reverse Index* gives the offsets for each vertex into this array. The *In2Out* provides a mapping from an edge in the *ReverseEdges* array to the corresponding edge in the *Edges* array. (b) presents the other data structures that are used. The dictionary provides a mapping from strings in the original RDF format to integers (a hash table). It also provides the reverse mapping (an array). However, the ids assigned during the dictionary phase are not the same as the ids assigned within the graph. Thus we need a way to map back and forth between them, which is accomplished with *Graph2RDF* (an array), and *RDF2Graph* (a hash table). The *Predicates* array gives the label for each of the edges. The i^{th} predicate corresponds to the i^{th} edge in the *Edges* array. The *In2Out* array is used to find the proper label for edges in the *Reverse Edges*.

selecting the triple pattern with the least amount of work. The least amount of work is determined by the following:

- If the subject of the triple pattern is constant, the outdegree of the subject.
- If the subject is a seen variable, the summation of outdegrees for the bindings.
- If the predicate is constant, the predicate multimap is queried to determine the number of edges with that predicate.
- If the predicate is a variable, the sum total of all edges with a predicate type that is recorded as valid for the variable.
- If the object is constant, the indegree of the object.
- If the object is a variable, the sum of all indegrees for all bindings.

B. Graph Joins

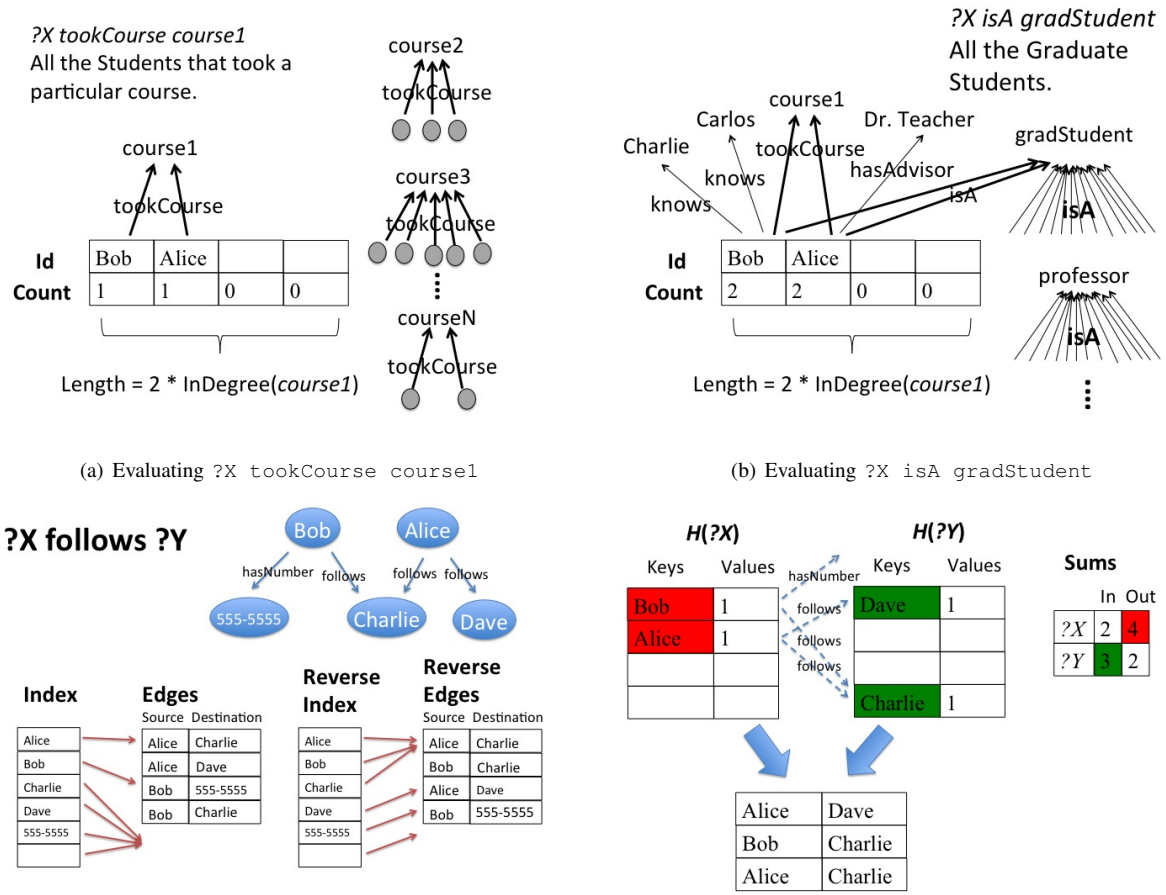
After sprinkling, we have a set of hash tables that contain a constrained list of variable bindings for each variable. We will refer to the set of bindings contained in the hash table for a variable *?X* as $H(?X)$. It is these hash tables that we now join together with graph joins. The graph joins output relational tables where each of the attributes are associated with a particular variable. We will refer to the values associated with variable *?X* in the table as $T(?X)$. Also, for a given row r in table T, $T(?X, r)$ is the particular binding for variable *?X* in row r.

During the Sprinkle phase, among the triple patterns evaluated are those with a single variable, and those bindings are already captured in the hash tables. As such, to create our final solution, we need only consider triple patterns with two or more variables, namely 1) *?S p ?O*, 2) *?S ?P o* or *s ?P ?O*, and 3) *?S ?P ?O*. The remainder of this section discusses the first two possibilities. Due to space constraints, we will not be discussing the third. We finish the section with a discussion on selecting the order of graph joins.

1) Graph Joins over ?S p ?O*:* Let us consider the first case when the subject and object are variables and the predicate is a constant. How the graph join is implemented is further subdivided by whether the variables have been processed or not by a previous join. There are three different possibilities: 1) neither *?S* nor *?O* has been processed by a previous join, 2) one has been processed by a previous join, or 3) both have been processed by a previous join.

No variables processed by previous join: When neither variable has been processed by a previous join, this means that both variables are represented by a unary relation that is stored in a hash table, i.e. the output of the Sprinkle phase. We have two options: iterate over the out-edges of the bindings currently in $H(?S)$ or iterate over the in-edges of the bindings currently in $H(?O)$. We know the total number of out- and in-edges as these numbers are updated after each sprinkle. Thus we can choose to expand out from the side with the least amount of work. Iterating over the in- or out-edges is proportional to the total number of in- or out-edges, thanks to the index structure of the CSR.

After deciding from which set of bindings to expand, we iterate over the edges, checking which have type *p*. For those edges that do have the proper predicate type, we then look up the object or subject of the edge (depending on if we



(a) Evaluating `?X tookCourse course1`

(b) Evaluating `?X isA gradStudent`

(c) The Graph and CSR

(d) Performing the Graph Join

Figure 3. (a) During the Sprinkle phase, a triple pattern is selected that has the least work either exploring in-edges, out-edges, or edges with a certain label. In this case, `?X tookCourse course1` is selected. Also, we expand out from the in-edges of *course1* as examining all edges with label *tookCourse* is much greater than the indegree of *course1*. Each subject id (Bob and Alice) is added to a hash table representing valid variable bindings for *?X*, and a counter for each binding is incremented. (b) For triple pattern `?X isA gradStudent`, since the hash table for variable *?X* is non-empty, Sprinkle SPARQL chooses to expand out from the variable bindings inside the table, rather than look at all the edges coming into *gradStudent* or all the edges with label *isA* Figures (c) and (d) give an example graph join: The query has the single triple pattern *?X follows ?Y*. The example graph is shown in Figure (c) with the corresponding CSR data structures. For readability, we present them with the original strings instead of integers. After the Sprinkle phase there are two hash tables for each of the variables, $H(?X)$ and $H(?Y)$. For display purposes, there are dashed edges between elements of the hash tables, but this information is actually contained in the CSR data structures. During the Sprinkle phase and also throughout the Join phase, sums of the total indegree and outdegree for each variable are kept. In this example, there are four out-edges in $H(?X)$ and three in-edges for variable $H(?Y)$. As such, the in-edges are explored first and we expand out from $H(?Y)$.

are iterating on out- or in-edges, respectively) in the hash table of the other variable. If it is present, then we create a 2-tuple with the matching subject and object and add it to an intermediate binary relational table. Algorithm 1 formalizes the notion. See Figure 3(c) and 3(d) for an example.

One variable processed by previous join When one of the two variables in an *?S p ?O*-type triple pattern has already been processed, then the bindings for the processed variable reside in an intermediate table. The graph join in this situation is similar to a classic hash join where one of the tables has been hashed. In our case, the hashed table is a unary relation of the non-joined variable.

The graph join for this case is similar to Algorithm 1. However, we limit our approach to scanning from the intermediate table and finding matches in the unary relation, rather than the other way around. There are potentially scenarios where it makes sense to hash the intermediate table and perform the join in the other direction, however we do not explore this issue in this paper.

Both variables processed by a previous join Our current join strategy, which will be discussed in Section IV-B3, has the limitation that all joins after the first one must

Algorithm 1. Graph Join for *?S p ?O* on Unary relations

procedure $A \leftarrow$ GRAPHJOIN_SO(S, p, O) ▷ S and O are both unary relations. p is a predicate type. A is the returned intermediate binary relation.
  $\sigma_S \leftarrow \sum_{s \in S} \delta^+(s)$
  $\sigma_O \leftarrow \sum_{o \in O} \delta^-(o)$
  if $\sigma_S < \sigma_O$ **then**
    for all $s \in S$ **do**
      for all $e \in out\_edges(s)$ **do**
        if $type(e) = p$ **then**
          if $e.object \in O$ **then**
            $A.add(s, e.object)$
          end if
        end if
      end for
    end for
  else
    for all $o \in O$ **do**
      for all $e \in in\_edges(o)$ **do**
        if $type(e) = p$ **then**
          if $e.subject \in S$ **then**
            $A.add(e.subject, o)$
          end if
        end if
      end for
    end for
  end if
  return A
end procedure

involve variables that have been seen in a previous join. As such, when processing an *?S p ?O*-type triple pattern where both variables have been processed in a previous join, we know they will be in the same intermediate table. We march through all the rows in the table, and validate that there is an edge with the appropriate type *p*. Again, to save work, for a given tuple with subject binding s and object binding o, we expand out from the node with the smallest out-degree or in-degree, respectively.

2) Graph Joins over ?S ?P o *or* s ?P ?O*:* We will now discuss how graph joins work on triple patterns of the form *?S ?P o* or *s ?P ?O*. Without loss of generality, we will focus on *?S ?P o*-type patterns. Similar to *?S p ?O*-type patterns, to evaluate *?S ?P o* we must consider whether or not the variables have been seen by a previous join. We consider when

1) neither *?S* nor *?P* has been processed by a previous join,
2) *?S* has been seen, but not *?P*,
3) *?P* has been seen, but not *?S*,
4) and when both have been processed by a previous join.

If neither *?S* nor *?P* has been seen, two choices exist: to expand out the out-edges of the bindings in $H(?S)$ or to consider the union of all edges that have an edge label contained in $H(?P)$. These statistics are known.

If we expand out from nodes in $H(?S)$, we consider each out-edge in $H(?S)$ and check if the out-edge has a predicate type that is found in $H(?P)$. If so we create a tuple for the edge in a binary relation, where one attribute is for *?S* and the other is for *?P*. The other direction is to iterate through all edges that have a type found in $H(?P)$. If the edge has a subject that is found in $H(?S)$, then the subject-predicate pair is added to the relation.

If *?S* has been seen but not *?P*, then it is the same as when neither has been seen before except that instead of using $H(?S)$, we instead draw from values in $T(?S)$. If *?S* has not been seen before but *?P* has, then we iterate over the in-edges of the constant object. For each in-edge of o, we find all corresponding entries in the intermediate table that has the same predicate type. For each one, we create a tuple. This particular case has not arisen very often, so we have not optimized it. We could either hash the in-edges of o or the intermediate table on the column associated with *?P* and do an operation similar to a hash join.

If both *?S* and *?P* have been processed before by a previous graph join, then we iterate through all rows of the intermediate table, and check and see that the *?S* attribute also has an edge of type equal to the *?P* attribute going to o.

3) Join Strategy using Graph Joins: Similar to [12], our join strategy focuses on sets of basic graph patterns having triple patterns that are all interrelated. Also similar to [12] we limit ourselves to execution plans that form a directed acyclic graph over this graph representation of the triple patterns. By this we mean we choose a triple pattern to evaluate first, and then every triple pattern thereafter must include a variable that has already been processed at least once. The general approach consists of following three steps, executed until all triple patterns have been evaluated: 1) select a triple pattern to evaluate, 2) perform the join and 3) update the statistics.

V. EVALUATION

We evaluate Sprinkle SPARQL on the Lehigh University Benchmark (LUBM) [7] and on an R-MAT [4] graph augmented with edge labels. We generated LUBM(8000), where 8000 is the number of universities. The number of triples in LUBM(8000) is approximately 1.1 billion, but with minimal RDFS [9] expands to ∼1.34 billion triples. We also generated triples by directly inferring that all graduate students are also students, resulting in a final total of ∼1.35 billion triples.

R-MAT, or a Recursive Model for Graph Mining, is an approach for generating graphs that have similar characteristics to real-world graphs such as social networks, the Internet

topology, and citation graphs. We created a graph with approximately 2^{30} edges and used the parameters specified by Graph500. We added edge labels, drawing from a set T where we vary $|T|$ from 1000 to 10,000. We assigned edge labels in a uniform random way from T.

For comparison, we evaluate against a Naïve approach. The Naïve approach works by giving an ordering to evaluate the triple patterns, subject to the same constraints as stated in Section IV-B3. Graph joins are then executed as specified by the order. As the Naïve method requires an ordering, in our experiments we attempt to evaluate all possible permutations, though in some instances we resort to sampling when the computation time becomes exorbitant.

For both data sets, we compare using two metrics:

- We compare the overall time for an equal number of processors. As the Naïve approach has $n!$ possible execution paths for n triple patterns, we directly compare against the best and worst times. Also, for relatively large n, we plot the distribution to give a notion of how likely each outcome is.
- We also examine the sum of intermediate result sizes: $\sum_{i=1}^{n} nv_i \cdot |r_i|$, where nv_i is the number of variables (number of columns) included in the i^{th} intermediate result and $|r_i|$ is the length of the result (number of rows), giving a feel for the amount of work that is pruned with *Sprinkle*.

A. Test Platforms

We used two platforms, a Cray XMT shared-memory supercomputer and an SGI Altix UV 10. The Cray system we use in this study has 128 processors and 1 TB of shared memory. The SGI Altix UV is another large-scale, shared memory system but, unlike the XMT, uses industry standards and commodity parts such as Xeon processors. The machine we use has 4 Xeon X7550 processors. Each processor has 8 cores/16 threads. Memory for the system is 0.5 TB.

B. LUBM

With experiments on LUBM, we first explore using just the Cray XMT. In the next subsection we examine relative scaling on both the Cray XMT and the SGI Altix UV on the more complicated queries, 2 and 9. We roughly categorize the LUBM queries in terms of their complexity.

Queries 1, 3, 5, 6, 10: These involve just one variable triple patterns. For 1, 3, 5, and 10 we present only the 2-processor results on the Cray XMT. Each of these queries have two triple patterns, one specifying the type of the variable `?X`, and the other specifying a constraint on `?X`. The trick for good performance on these queries is to select the latter triple, the constraint. Evaluating first the triple specifying the type of `?X` invariably leads to many matches, almost all of which are discarded. Sprinkle SPARQL has no troubles in selecting the constraint triple first, as the degree on the specified object is quite small in all cases. Table I compares times of Sprinkle SPARQL vs. Naïve for all the LUBM queries we examined.

Query	Sprinkle SPARQL	Naïve Best	Naïve Worst	Num Procs
1	0.0669	0.0670	45.72	2
2	5.2017	5.5653	6007	128
3	0.0690	0.0663	61.94	2
4	0.3165	0.1452	540.8	2
5	0.0298	0.0102	184.0	2
6	4.4001	0.5622	0.5622	128
7	0.1558	0.1277	1427	2
8	0.3625	0.1924	483.2	2
9	13.7616	7.3303	66.29	128
10	0.0785	0.0687	174.6	2

Table I
THIS TABLE COMPARES TIMES IN SECONDS OF SPRINKLE SPARQL VERSUS THE BEST AND WORST NAÏVE TIMES. HOW MANY PROCESSORS WERE USED IN THE RUNS IS ALSO SPECIFIED.

Query	Number of Results	Sprinkle SPARQL	Naïve Best	Percent Change	Naïve Worst
1	4	4	8	100	20,157,123
2	2,528	60,798,953	161,272,120	165	5,539,651,344
3	6	6	12	100	64,478,867
4	34	306	381	19.7	986,917,868
5	719	719	1438	100	89,318,851
6	83,557,706	83,557,706	83,557,706	0	83,557,706
7	67	132	406	208	1,383,110,222
8	7,790	38,950	55,640	42.8	965,241,544
9	2,178,420	307,234,069	326,065,015	6.13	3,233,792,132
10	4	4	8	100	83,557,710

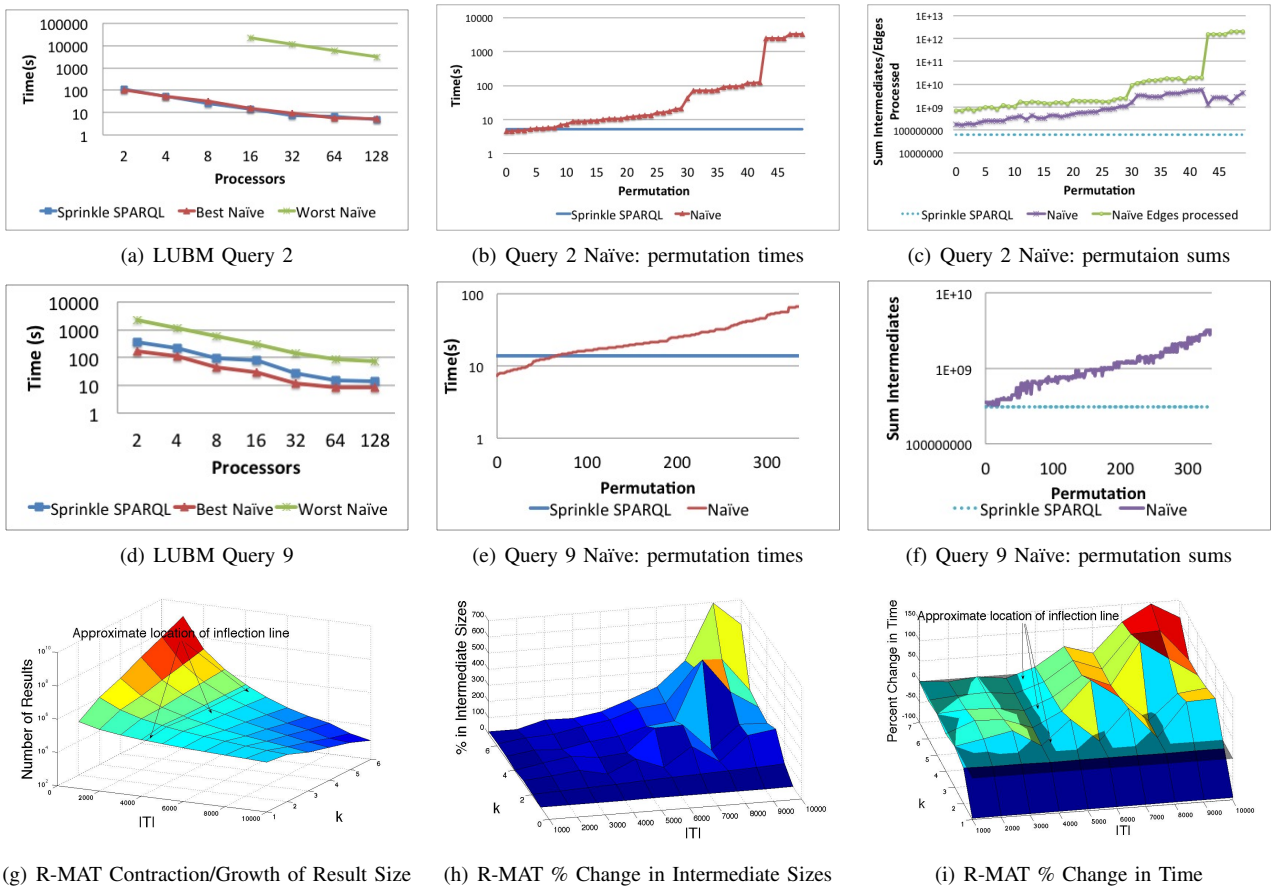
Table II
THIS TABLE COMPARES THE SUMMATION OF INTERMEDIATE JOIN SIZES OF SPRINKLE SPARQL VERSUS THE BEST AND WORST SUMS VIA THE NAÏVE METHOD. THE *Percent Change* COLUMN SPECIFIES THE PERCENTAGE CHANGE FROM THE SUM OF SPRINKLE SPARQL TO THE BEST SUM OF NAÏVE.

Table II compares intermediate sizes during the join phase. However, we note that Sprinkle SPARQL does not perform any joins when no two variable triple patterns are present. The result is extracted from the variable's hash table. As such, we report the final result size in Table II.

Query 6 asks for all Students. Query 6 does reveal a small weakness of Sprinkle SPARQL. With only one triple pattern, the ordering for both methods is the same, so Sprinkle adds overhead but reaps no benefit. This weakness can easily be averted by executing the Naïve method in such a situation.

Queries 4, 7, and 8 involve two variables, but are still relatively simple. Table I shows that Sprinkle SPARQL results in times that are competitive with the best permutation of Naïve. However, there are many more possible permutations for these queries, and Figure 5 shows the range of times. The times vary over many orders of magnitude.

Queries 2 and 9 are the more complicated LUBM queries and have at their core a triangle. For query 2, the



(a) LUBM Query 2 (b) Query 2 Naïve: permutation times (c) Query 2 Naïve: permutaion sums

(d) LUBM Query 9 (e) Query 9 Naïve: permutation times (f) Query 9 Naïve: permutation sums

(g) R-MAT Contraction/Growth of Result Size (h) R-MAT % Change in Intermediate Sizes (i) R-MAT % Change in Time

Figure 4. For LUBM query 2, Figure (a) compares Sprinkle SPARQL against the best and worst permutations of the Naïve approach on the XMT. Figure (b) shows the range of times the permutations took with 128 procs while Figure (c) examines the sum of intermediate sizes across permutations (and also processed edges). Figure (d)-(f) are for query 9. (g) On the R-MAT graph, shows the growth in query result size with increasing k for $|T| < 4210$ and contraction for $|T| > 4210$. (h) Shows % change in sum of intermediate sizes from Sprinkle SPARQL to Naïve. For all $|T|$ and number of graph joins, k, Naïve produces intermediate results that are greater than or equal to Sprinkle SPARQL. (i) Displays the % change in run time from Naïve to Sprinkle SPARQL. Transparent plane marks 0% change. Sprinkle SPARQL does best in the region $|T| > 4210$ and where $k > 1$.

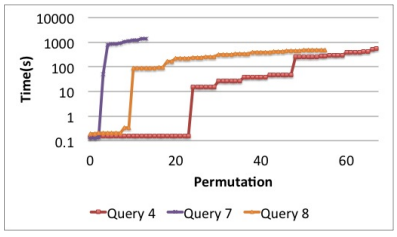


Figure 5. Sorted times for permutations of LUBM Queries 4, 7, and 8 using the Naïve method.

naïve implementation has a large variance, ranging from 5.56 to 6007 seconds. Due to some permutations taking an inordinate amounts of time, we did not run every possible permutation. We ran the Naïve approach on 50 different randomly sampled permutations. The runs with 128 processors, sorted by time, are presented in Figure 4(b). We also present the summation of intermediate sizes for each query in Figure 4(c). However, the determining factor in run time appears to be the number of processed edges.

Some orderings amass large numbers of high-degree vertices, resulting in over a trillion edges processed! Of the sample of 50, we found the best and worst permutations for the naïve implementation and ran 2 to 128 processors. Sprinkle SPARQL and the best permutation are about even.

For query 9, Sprinkle SPARQL didn't do as well as the best Naïve permutation. From Table II we note the difference in intermediate sizes is only off by about 6%. As such, Sprinkle SPARQL couldn't overcome its extra overhead and it took roughly twice the time of the best Naïve permutation. However, Sprinkle SPARQL did select the best ordering.

C. Relative Scalability

In Figure 6 we look at the times and relative scalability of LUBM queries 2 and 9, the more complicated of the LUBM queries. For LUBM query 2, the UV showed a relative speedup of 8.30x over 32 threads. The XMT exhibited a speedup of 26.92x over 64 processors and 26.62x over 128

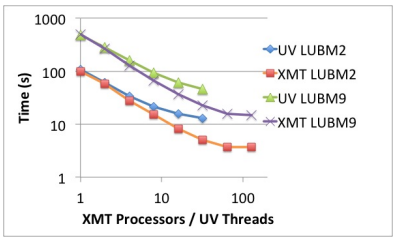


Figure 6. This figure shows the relative scalability of LUBM queries 2 and 9 on LUBM(8000) for both a Cray XMT and an SGI Altix UV 10.

processors. For LUBM query 9, the UV had a relative speedup of 10.70x over 32 threads. The XMT had a speedup of 32.16x for 64 processors and 34.17x for 128 processors. We believe the decrease in scalability largely arises due to the decreasing amount of parallelism as the Sprinkle phase or Join phase progresses. As explained earlier, triple patterns are evaluated sequentially. Some triple patterns have very little work, so not all processing power can be used efficiently in these circumstances. Future work will look at evaluating multiple triple patterns in tandem.

D. R-MAT

As outlined previously, we generated an R-MAT graph with a billion edges and then added edge types in a uniform random way. We selected sequences of that pattern to be the focus of our study. More precisely, we evaluated path queries of length k: $?x_1\ p_1\ ?x_2\ p_2\ ?x_3$ ... $?x_k\ p_k\ ?x_{k+1}$.

We varied $|T|$ from 1000 to 10,000 and k from 1 to 7 and found a curious result as seen in Figure 4(g). We found that for small $|T|$, the size of the query result grows with increasing k. For large $|T|$, the size of the query contracts with increasing k. Experimentally, we found the inflection line to be around $|T| = 4210$. We will next discuss the impact that this line has on the performance.

Figure 4(h) shows the percentage change in the summation of intermediate results from Sprinkle SPARQL to the Naïve approach. Positive numbers indicate larger intermediate sizes for the Naïve approach relative to Sprinkle SPARQL. For one graph join, the size is the same for both methods. For every other data point, Sprinkle SPARQL reduces the sum. The relative increase ranges from 4.7% to 696%. The effect is dramatic for large $|T|$ and large k.

Savings in intermediate sizes translates into improved performance, shown in Figure 4(i). The figure shows the percent change in time from Sprinkle SPARQL to Naïve. There is a transparent plane marking 0%. Again, similar to LUBM query 6, Sprinkle SPARQL performs significantly worse than the naïve approach for single graph joins. However, for $k > 2$, the performance difference ranges from -16.9% to 145%. The region where Sprinkle SPARQL performs the best greater than the inflection line, $|T| > 4210$.

VI. Conclusions

We have shown that Sprinkle SPARQL fulfills the two desiderata we outlined earlier: 1) it removes invalid variable bindings with low cost Sprinkle operations before expensive join operations, and 2) it selects a near optimal path for executing the joins. For all of our experimental studies, Sprinkle SPARQL does select the optimal execution plan. While in some cases the best permutation for the Naïve approach fared better than Sprinkle SPARQL, our algorithm presents an efficient method to discover that permutation.

Acknowledgments.: This work was partially funded under the Center for Adaptive Supercomputing Software – Multithreaded Architectures (CASS-MT) at the Dept. of Energy's Pacific Northwest National Laboratory. Pacific Northwest National Laboratory is operated by Battelle Memorial Institute under Contract DE-ACO6-76RL01830.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. This work was partially funded by Sandia's University Part-time program, and is part of Eric Goodman's dissertation.

References

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Sw-store: a vertically partitioned dbms for semantic web data management. *The VLDB Journal*, 18(2):385–406, Apr. 2009.

[2] K. Anyanwu, H. Kim, and P. Ravindra. Algebraic optimization for processing graph pattern queries in the cloud. *Internet Computing, IEEE*, PP(99):1, 2012.

[3] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. S. T. Sterling, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems, peter kogge, editor and study lead, 2008.

[4] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In M. W. Berry, U. Dayal, C. Kamath, and D. B. Skillicorn, editors, *SDM*. SIAM, 2004.

[5] E. Goodman, M. N. Lemaster, and E. Jimenez. Scalable hashing for shared memory supercomputers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 41:1–41:11, New York, NY, USA, 2011. ACM.

[6] E. L. Goodman, D. J. Haglin, C. Scherrer, D. Chavarría-Miranda, J. Mogill, and J. Feo. Hashing strategies for the cray xmt. In *Workshop on Multithreaded Architectures and Applications*, April 2010.

[7] Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

[8] M. Husain, J. McGlothlin, M. M. Masud, L. Khan, and B. M. Thuraisingham. Heuristics-based query processing for large rdf graphs using cloud computing. *IEEE Trans. on Knowl. and Data Eng.*, 23(9):1312–1327, Sept. 2011.

[9] S. Muñoz, J. Pérez, and C. Gutierrez. Simple and Efficient Minimal RDFS. *Web Semantics: Science, Services and Agents on the World Wide Web*, 7(3):220–234, Sept. 2009.

[10] T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *The VLDB Journal*, 19(1):91–113, Feb. 2010.

[11] T. Neumann and G. Weikum. x-rdf-3x: fast querying, high update rates, and consistency for rdf databases. *Proc. VLDB Endow.*, 3(1-2):256–263, Sept. 2010.

[12] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. Sparql basic graph pattern optimization using selectivity estimation. *Proceeding of the 17th international conference on World Wide Web WWW 08*, page 595, 2008.

[13] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gstore: answering sparql queries via subgraph matching. *Proc. VLDB Endow.*, 4(8):482–493, May 2011.